算法模板与思路

一、时空间复杂度

一般ACM或者笔试题的时间限制是1秒或2秒，`C++`一般 1秒能计算 $10^7 - 10^8$ 次，在这种情况下，`C++` 代码中的操作次数控制在 10^7 为最佳.

在不同数据范围下，代码的时间复杂度和算法的选择技巧如下：

数据范围	时间复杂度	算法
$n \leq 30$	$O(2^n)$	dfs+剪枝, 状态压缩dp
$n \leq 10^2$	$O(n^3)$	floyd, dp, 高斯消元
$n\leq 10^3$	$O(n^2)$, $O(n^2logn)$	dp, 二分, Dijkstra,Prim, Bellman-Ford
$n\leq10^4$	$O(n*\sqrt n)$	块状链表, 分快, 莫队

数据范围	时间复杂度	算法
$n\leq10^5$	$O(n*logn)$	sort, heap, set/map, 二分, 拓扑排序
		Dijkstra_heap, Prim_heap, Kruskal, 整体二分
		spfa, CDQ分治, 求凸包, 求半平面交
		后缀数组, 树链剖分, 线段树, 树状数组, 动态树
$n\leq 10^6$	$O(n)$	单调队列,hash,双指针,KMP,并查集,AC自动机
$n\leq 10^6$	常数小的$O(n*logn)$	sort、树状数组、heap、dijkstra、spfa
$n \leq 10^7$	$O(n)$	双指针、kmp、AC自动机、线性筛素数
$n \leq 10^9$	$O(\sqrt n)$	判断质数
$n \leq 10^{18}$	$O(logn)$	最大公约数，快速幂，数位DP
$k \leq 10^{1000}$	$O((logk)^2)$, k表位数	高精度加减乘除
$k \leq 10^{10^5}$	$O(logk \times logk)$	高精度加减、FFT/NTT

二、基础算法

2.1 排序

2.1.1 快速排序

快速排序

```
void quick_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }

    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}
```

```
def quick_sort(nums, l, r):
```

```python
        if l >= r: return

        i, j, x = l-1, r+1, nums[l + r >> 1]
        while i < j:
            while True:
                i += 1
                if nums[i] >= x: break
            while True:
                j -= 1
                if nums[j] <= x: break
            if i < j:
                nums[i], nums[j] = nums[j], nums[i]
        quick_sort(nums, l, j)
        quick_sort(nums, j+1, r)
```

2.1.2 归并排序

[归并排序]

```cpp
int N = 100001;
int q[N], tmp[N];

void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];

    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

```python
tmp = [0 for _ in range(len(nums))]

def merge_sort(nums, l, r):
    if l >= r: return
    mid = l + r >> 1
    merge_sort(nums, l, mid)
    merge_sort(nums, mid+1, r)

    i, j, k = l, mid+1, 0
    while i <= mid and j <= r:
        if nums[i] <= nums[j]:
            tmp[k] = nums[i]
```

```
            i += 1
        else:
            tmp[k] = nums[j]
            j += 1
        k += 1

    while i <= mid:
        tmp[k] = nums[i]
        i += 1
        k += 1
    while j <= r:
        tmp[k] = nums[j]
        j += 1
        k += 1
    for i in range(l, r+1):
        nums[i] = tmp[i-l]
```

2.2 二分

> 作用： 对单调序列的查找可以优化时间复杂度，$O(n) \rightarrow O(log n)$ 。

2.2.1 整数二分

[数的范围]

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
// 找左边界
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) l = mid + 1;    // check()判断mid是否满足性质
        else r = mid;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
// 找右边界
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) r = mid - 1;
        else l = mid;
    }
    return l;
}
```

```
def check(mid):
```

```
    # 检测mid满足某种性质
    pass
def bsearch_1(l, r):
    while l < r:
        mid = l + r >> 1
        if check(mid): l = mid + 1
        else: r = mid
    return l
def bsearch_2(l, r):
    while l < r:
        mid = l + r +  1 >> 1
        if check(mid): r = mid - 1
        else: l = mid
    return l
```

2.2.2 浮点数二分模板

[数的三次方]()

```
bool check(double x) {/* ... */} // 检查x是否满足某种性质

double bsearch_3(double l, double r)
{
    // eps 表示精度，如保留4位小数用1e-6
    const double eps = 1e-6;
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
```

2.3 前缀和与差分

> 对数组中一段序列或矩阵中一个小矩阵**求和**或**同时加上一个数**，能够优化时间复杂度。$O(n) \rightarrow O(1)$

2.3.1 一维前缀和

[前缀和]()

```
# 定义： a[N] 为给定数组，S[N] 为 a[N] 的前缀和
# 计算：
S[i] = a[1] + a[2] + a[3] + ... + a[i]
a[l] + ... + a[r] = S[r] - S[l-1]
```

2.3.2 二维前缀和

子矩阵的和

```
# 定义： a[N][N] 为二维矩阵，s[N][N] 为 a[N][N] 的前缀和
# 前缀和定义： s[i][j] 为 a[i][j] 格子左上部分所有元素的和
# 1. 如何求 s[i, j]?
s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1, j-1] + a[i][j]

# 2. 如何求以（x1, y1)为左上角，(x2, y2)为右下角的子矩阵和？
res = s[x2][y2] - s[x2][y1-1] - s[x1-1][y2] + s[x1-1][y1-1]
```

2.3.3 一维差分

差分

```
// 给定a[1],a[2],...,a[n], 构造差分数组b[N],使得 a[i] = b[1] + b[2] + ... + b[i]
// 核心操作： 将 a[L~R] 全部加上 C，等价于 b[L] += C, b[R+1] -= C, 其中 L <= R.
const int N=1001;
int a[N], b[N];
void insert(int l, int r, int c)
{
    b[l] += c;
    b[r+1] -= c;
}
```

2.3.4 二维差分

差分矩阵

```
// 原矩阵 a[N][N]; 差分矩阵 b[N][N]
// 满足性质： a[i][j] 是 b[i][j]左上角所有元素的和
// 对原矩阵的操作： 以 a[x1][y1] 为左上角， a[x2][y2] 为右下角的矩阵中所有元素分别加 C
// 等价于差分矩阵操作：
b[x1][y1] += c;
b[x2+1][y1] -= c;
b[x1][y2+1] -= c;
b[x2+1][y2+1] += c;
```

2.4 双指针

> 优化。$O(n^2) \rightarrow O(n)$

最长连续不重复自序列

数组元素的目标和

判断自序列

```
// i:左指针; j:右指针
for (int i=0, j=0; j<n; j++)
{
    while(check(i, j)) i++;
    // 具体问题逻辑
}
// 常见问题分类:
//  (1) 对于一个序列，用两个指针维护一段区间
//  (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作
```

2.5 位运算

[二进制中1的个数]

```
// 求 n 的二进制表示中第 k 位数字
// 应用: 输出二进制表示
n >> k & 1;

// 返回 n 二进制表示中最后一位 1 表示的大小
// 应用: 统计二进制中 1 的个数
int lowbit(n)
{
    return n & -n;
    // -n 等价于 n取反加1
    // return n & (~n + 1);
}

// 消除数字 n 的二进制表示中的最后一个 1
n & n-1;

// 异或的性质
n ^ n = 0;
n ^ 0 = n;
a ^ b ^ a = a;
```

2.6 离散化

```
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素

// 二分求出x对应的离散化的值
int find(int x) // 找到第一个大于等于x的位置
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
```

```
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1; // 映射到1, 2, ...n
}
```

2.7 区间合并

```
using namespace std;
typedef pair<int, int> PII;

// 将所有存在交集的区间合并
void merge(vector<PII> &nums)
{
    vector<PII> res;
    sort(nums.begin(), nums.end());

    // 默认至少有一个区间, 可以减少边界处理
    int st = nums[0].first, ed = nums[0].second;
    for(auto pii: nums)
    {
        if(ed < pii.first)
        {
            ans.push_back({st, ed});
            st = pii.first, ed = pii.second;
        }
        else ed = max(ed, pii.second);
    }
    ans.push_back({st, ed});
    // 这个位置的赋值需要引用传参才有效
    nums = ans;
}
```

三、数据结构

3.1 链表

3.1.1 单链表

```
// head存储链表头, e[]存储节点的值, ne[]存储节点的next指针, idx表示当前用到了哪个节点
int head, e[N], ne[N], idx;

// 初始化
void init()
{
    head = -1;
    idx = 0;
```

```
}

// 在链表头插入一个数a
void insert(int a)
{
    e[idx] = a, ne[idx] = head, head = idx ++ ;
}

// 将头结点删除，需要保证头结点存在
void remove()
{
    head = ne[head];
}
```

3.1.2 双链表

3.2 栈

3.2.1 模拟栈

3.2.2 单调栈

3.3 队列

3.3.1 普通队列

3.3.2 循环队列

3.3.3 单调队列

3.4 KMP

3.5 Trie树

```
int son[N][26], cnt[N], idx;
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
```

```
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

3.6 并查集

3.7 堆

3.8 哈希

3.8.1 一般哈希

3.8.2 字符串哈希

3.9 C++ STL

```
vector, 变长数组，倍增的思想
    size()  返回元素个数
    empty()  返回是否为空
    clear()  清空
    front()/back()
    push_back()/pop_back()
    begin()/end()
    []
    支持比较运算，按字典序

pair<int, int>
    first, 第一个元素
    second, 第二个元素
    支持比较运算，以first为第一关键字，以second为第二关键字（字典序）
```

```
string，字符串
    size()/length()  返回字符串长度
    empty()
    clear()
    substr(起始下标，(子串长度))  返回子串
    c_str()  返回字符串所在字符数组的起始地址

queue, 队列
    size()
    empty()
    push()  向队尾插入一个元素
    front()  返回队头元素
    back()  返回队尾元素
    pop()  弹出队头元素

priority_queue, 优先队列，默认是大根堆
    size()
    empty()
    push()  插入一个元素
    top()  返回堆顶元素
    pop()  弹出堆顶元素
    定义成小根堆的方式：priority_queue<int, vector<int>, greater<int>> q;

stack, 栈
    size()
    empty()
    push()  向栈顶插入一个元素
    top()  返回栈顶元素
    pop()  弹出栈顶元素

deque, 双端队列
    size()
    empty()
    clear()
    front()/back()
    push_back()/pop_back()
    push_front()/pop_front()
    begin()/end()
    []

set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列
    size()
    empty()
    clear()
    begin()/end()
    ++, -- 返回前驱和后继，时间复杂度 O(logn)

    set/multiset
        insert()  插入一个数
        find()  查找一个数
        count()  返回某一个数的个数
        erase()
            (1) 输入是一个数x，删除所有x   O(k + logn)
            (2) 输入一个迭代器，删除这个迭代器
        lower_bound()/upper_bound()
```

```
            lower_bound(x)  返回大于等于x的最小的数的迭代器
            upper_bound(x)  返回大于x的最小的数的迭代器
    map/multimap
        insert()  插入的数是一个pair
        erase()  输入的参数是pair或者迭代器
        find()
        []  注意multimap不支持此操作。 时间复杂度是 O(logn)
        lower_bound()/upper_bound()

unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
    和上面类似，增删改查的时间复杂度是 O(1)
    不支持 lower_bound()/upper_bound()， 迭代器的++，--

bitset, 压位
    bitset<10000> s;
    ~, &, |, ^
    >>, <<
    ==, !=
    []

    count()  返回有多少个1

    any()  判断是否至少有一个1
    none()  判断是否全为0

    set()  把所有位置成1
    set(k, v)  将第k位变成v
    reset()  把所有位变成0
    flip()  等价于~
    flip(k) 把第k位取反
```

四、搜索与图论

4.1 DFS与BFS

```
queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
q.push(1);

while (q.size())
{
    int t = q.front();
    q.pop();

    for (int i = h[t]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true; // 表示点j已经被遍历过
            q.push(j);
        }
```

```
    }
}
```

4.2 树与图的遍历：拓扑排序

4.3 最短路

4.4 最小生成树

4.5 二分图：染色法、匈牙利算法

五、贪心

六、动态规划

6.1 背包问题

6.1.1 01背包问题

6.1.2 完全背包问题

6.1.3 多重背包问题

6.1.4 二进制优化多重背包

6.1.5 分组背包问题

七、数学知识